WORKFLOW AND PARALLELEXTENSIONS IN .NET FRAMEWORK 4

Ling Wo, Cristina Manu

Parallel Computing Platform
Microsoft Corporation

TABLE OF CONTENTS

INTRODUCTION

In the .NET Framework 4, System.Threading.Tasks.Parallel is a new class in mscorlib.dll that supports structured parallelism through multithreading. It contains a set of overloads that can be thought as parallel replacements for the existing sequential "foreach" and "for" looping constructs that exist in languages like C#, Visual Basic, and F#. Together with the other types in System.Threading.Tasks, Parallel LINQ (PLINQ), and a significant number of thread-safe collections and synchronization primitives new to .NET 4, the Parallel class offers strong support for the development of managed, multi-threaded applications. All of these types in aggregate are commonly referred to as "Parallel Extensions to the .NET Framework 4", or "Parallel Extensions" for short.

Windows Workflow Foundation in .NET 4 (abbreviated WF4), enables users to develop workflow services and workflow applications. Workflows can be built programmatically or using a visual designer, permit the use of custom control flow (e.g. Flowcharts), and can be persisted to durable storage when they are idle awaiting input. When defining workflows, it is convenient to express multiple paths of execution using the Parallel and ParallelForEach activities. The WF runtime executes an instance of a workflow in a single-threaded fashion, which means that the behavior of WF's Parallel activity is fundamentally different than that of System.Threading.Tasks.Parallel.

Despite the similarity in naming, System.Threading.Tasks.Parallel and the System.Activities.Statements.Parallel* activities in WF4 are largely orthogonal in the scenarios they address. However, WF4 activities and the new parallel programming types in .NET 4 can be used together to great advantage. To this end, this document has two primary goals:

1. To explain the behavioral differences between the Parallel activities and System.Threading.Tasks.Parallel.* APIs as well as differences for when each is more appropriate to use.
2. To demonstrate how Parallel Extensions can be integrated into WF4 in order to support development of multithreaded workflow applications.

WORKFLOW PARALLEL ACTIVITY VS. SYSTEM.THREADING.TASKS.PARALLEL.INVOKE

DESCRIPTION

Both the System.Activities.Statements.Parallel activity and the System.Threading.Tasks.Parallel.Invoke method represent compositions of other units of computation. When started, they will not complete until all the child operations have completed, either successfully or due to early termination, such as through cancellation.

PARALLEL ACTIVITY

When building workflows, there are three scenarios in which the Parallel activity is useful.

The first scenario is a workflow that expects inputs in a certain pattern. For example, you may be implementing a stateful web service that expects inputs (client invocations of web service operations) in a certain order.

Here is a sketch of such a service, where A, B, C, and D are activities that represent distinct service operations. The use of the Parallel activity allows the user to easily express that the operations may be invoked in one of two

orderings: A, B, C, D or A, C, B, D. The Parallel activity completes only when both B and C are complete, but those two operations can occur in either order.

```
<Sequence>
        <A />
        <Parallel>
                <B />
                <C />
        </Parallel>
        <D />
</Sequence>
```

The second scenario for the Parallel activity involves the coordination of asynchronous work. Suppose, in our stateful web service, that after service operation A is called, we need to write some information to a database, and also invoke an operation on (send a message to) another web service. Use of the Parallel activity, along with activities (in this example, System.ServiceModel.Activities.Send and a custom InsertToDb activity) that implement WF4's AsyncCodeActivity pattern allow these I/O operations to be executed concurrently without blocking the thread used by the WF runtime to execute activities in the workflow instance. Please consult the WF4 documentation for additional information about asynchronous activities.

```
<Sequence>
        <A />
        <Parallel>
                <Send … />
                <InsertToDb />
        </Parallel>
        <Parallel>
                <B />
                <C />
        </Parallel>
        <D />
</Sequence>
```

The third scenario is a variant of the second, the difference being that the asynchronous activities present in the branches of the Parallel represent expensive computation rather than I/O. Later in this paper we will see how to utilize System.Threading.Tasks to accomplish this scenario.

In order to create an instance of a Parallel activity the Branches need to be defined; they represent the child activities for the Parallel. When the Parallel activity executes, it schedules its child activities via the WF runtime. The execution of Parallel, like all activities, is single threaded: the WF runtime utilizes one thread per workflow instance but, in the case of Parallel, achieves interleaving across branches since activities can block awaiting data (or the result of an asynchronous operation) without holding onto the thread.

The below sample illustrates the usage of the Parallel activity. The Parallel activity is wrapped in a Sequence activity in order to demonstrate the fact that the Parallel activity will not be completed until all its children activities are completed. The example uses a Delay activity to demonstrate the asynchronous execution of Parallel activity's branches.

```
Activity workflow = new Parallel
            {
                Branches =
                {
                    new Sequence
                    {
                        Activities =
                        {
                            new WriteLine{ Text = "one"},
                            new Delay{ Duration = TimeSpan.FromSeconds(5)},
                            new WriteLine{ Text = "two"}
                        }
```

```
                    },
                    new Sequence
                    {
                         Activities =
                        {
                            new WriteLine{ Text = "three"},
                            new Delay{ Duration = TimeSpan.FromSeconds(5)},
                            new WriteLine{ Text = "four"}
                        }
                    }
                }
            };

WorkflowInvoker.Invoke(workflow);
```

SYSTEM.THREADING.TASKS.PARALLEL.INVOKE

The System.Threading.Tasks.Parallel class provides several static methods for executing operations in parallel. This includes an Invoke method, which is meant to serve as a parallelized replacement for a sequential region of statements. For example, consider the sequential sequence of statements:

```
Console.WriteLine("Branch1:: ManagedThreadId is :" " +
Thread.CurrentThread.ManagedThreadId);
Console.WriteLine("Branch2:: ManagedThreadId is :" "+
Thread.CurrentThread.ManagedThreadId);
```

We can invoke these in parallel with code such as:

```
Action[] actions = new Action[2]{
()=>{Console.WriteLine("Branch1:: ManagedThreadId is :" " +
Thread.CurrentThread.ManagedThreadId);},
()=>{Console.WriteLine("Branch2:: ManagedThreadId is :" "+
Thread.CurrentThread.ManagedThreadId);}};
Parallel.Invoke(actions);
```

Because the execution of Parallel.Invoke is multithreaded, the user should be careful in the application's logic when accessing shared data. By default, which and how many threads utilized are controlled by a combination of logic from ThreadPool and Parallel.Invoke. The algorithm is highly tuned in order to provide quality utilization of machine resources.

Parallel.Invoke enables varying degrees of control over how the actions provided to it are executed. For example, the user can set a maximum degree of parallelism through use of the ParallelOptions type and its MaxDegreeOfParallelism property. This will ensure that no more than the specified number of tasks will be run simultaneously during Paralle.Invoke's execution.

As was true in case of the Parallel activity, the order of execution of the actions to be processed in parallel is not guaranteed and the user should not rely on it. Parallel.Invoke will not return until all until all the given actions are either completed or cancelled.

Please remark that, even if the scenario is similar with the one described for Parallel activity sample, in this case the collection that the child actions will operate on needs to be thread safe, condition not necessary in Parallel activity case.

```
public static void ParallelInvokeSample(IProducerConsumerCollection<string > collection)
{
            Action[] actions = new Action[3]{
                ()=>{ collection.TryAdd("*** Activity 1; Current Time:" +
System.DateTime.Now.ToLongTimeString() + "; ThreadId " +
Thread.CurrentThread.ManagedThreadId);},
                ()=>{ collection.TryAdd("*** Activity 2; Current Time:" +
System.DateTime.Now.ToLongTimeString() + "; ThreadId " +
Thread.CurrentThread.ManagedThreadId);},
                ()=>{ collection.TryAdd("*** Activity 3; Current Time:" +
System.DateTime.Now.ToLongTimeString() + "; ThreadId " +
Thread.CurrentThread.ManagedThreadId);}
            };

            Parallel.Invoke(actions);
}

. . .

ConcurrentQueue<string> myCollection = new ConcurrentQueue<string>();
ParallelInvokeSample(myCollection);
myCollection.Enqueue("--- LAST ELEMENT ---");
```

CANCELLATION

PARALLEL ACTIVITY

Cancellation is often a normal part of workflow execution. To support scenarios where cancellation is expected, the Parallel activity has a CompletionCondition that is evaluated after the completion of each branch; if the CompletionCondition is true the pending branches are cancelled.

Note that even if the CompletionCondition is true from the beginning, the Parallel activity will not complete until a child activity completes. This is different from the Parallel.Invoke cancellation behavior, which will be shown later.

In the below sample the loop will exit if a specific string is found in the collection.

```
public static Activity CreateWf(int delayTimeInSec, ICollection<string> myCollection)
        {
            //when this item is added to the collection the parallel Activity will end
            string itemToAdd = "!!! Common !!!";

            Variable<ICollection<string>> collection = new
            Variable<ICollection<string>>("MyCollection", myCollection);

            Sequence addWithDelay = new Sequence()
            {
                Activities =
                {
                    new Delay { Duration = TimeSpan.FromSeconds(delayTimeInSec)},
                    new AddToCollection<string>
                    {
                        Collection=collection,
                         Item= new InArgument<String>(e => "*** Activity 2; Current Time:"
                         + System.DateTime.Now.ToLongTimeString() + "; ThreadId " +
                         Thread.CurrentThread.ManagedThreadId)
                    },
                }
            };
```

```
            var workflow = new Parallel()
            {
                Variables = { collection },
                CompletionCondition = new ExistInCollection<string>(){ Collection =
                collection, Item = itemToAdd },
                Branches =
                        {
                            new AddToCollection<string>
                            {
                                Collection=collection,
                                Item= new InArgument<String>(e => "*** Activity 1; Current
                                Time:" + System.DateTime.Now.ToLongTimeString() + ";
                                ThreadId " + Thread.CurrentThread.ManagedThreadId)
                            },
                            addWithDelay,

                           //add the common string; the CompletionCondition will be true
                         this branch finishes
                            //the delayed action will not be run
                            new AddToCollection<string>
                            {
                                Collection=collection,
                                Item= new InArgument<String>(e => itemToAdd )
                            }
                        }
            };

            return workflow;
        }
List<string> myCollection = new List<string>();
WorkflowInvoker.Invoke(CreateWf (5/*delayed time*/, myCollection));
```

Cancellation can also occur as a result of an operation on the host of a workflow instance, e.g. the WorkflowApplication.Cancel method.

SYSTEM.THREADING.TASKS.PARALLEL.INVOKE

The .NET Framework 4 includes two new types in the System.Threading namespace specifically targeted at canceling large swaths of asynchronous work: CancellationToken and CancellationTokenSource. More information on these types is available at http://blogs.msdn.com/9635790.aspx; in short, however, CancellationToken is a small type that serves as a notification mechanism for cancellation being requested, and CancellationTokenSource is a type that both creates a CancellationToken and requests its cancellation. All of the new parallelization support in .NET 4 utilizes CancellationToken for its cancellation needs: this includes Parallel.Invoke.

In the specific case of the Parallel.Invoke, a CancellationToken is passed to the Parallel.Invoke call through a ParallelOptions instance. The following sample shows how the execution of the Parallel.Invoke can be cancelled.

```
CancellationTokenSource cts = new CancellationTokenSource();
Action[] actions = new Action[3]
            {
                ()=>{ collection.TryAdd("*** Activity 1; Current Time:" +
        System.DateTime.Now.ToLongTimeString() + "; ThreadId " +
        Thread.CurrentThread.ManagedThreadId);},
                ()=>{ cts.Cancel();},
                ()=>{ collection.TryAdd("*** Activity 3; Current Time:" +
        System.DateTime.Now.ToLongTimeString() + "; ThreadId " +
        Thread.CurrentThread.ManagedThreadId);}
            };
ParallelOptions options = new ParallelOptions() { CancellationToken = cts.Token };
try
{
```

```
        Parallel.Invoke(options, actions);
 }
catch (Exception e)
{
        Console.WriteLine("Exception: {0}::{1}", e.GetType(), e.Message );
}
```

It is important to mention the following differences between the Parallel.Invoke cancellation and ParallelActivity cancellation.

1. With Parallel.Invoke, if the token is cancelled from the beginning, no Action will run. As mentioned previously, with a Parallel activity, even if the CompletionCondition is true when the Parallel begins executing, one branch will still be completed before cancellation of other branches occurs.
2. When cancellation is observed, the Parallel activity will not throw an exception. In contrast, Parallel.Invoke will result in a System.OperationCanceledException being thrown if canceled to signify to the caller that not all of the requested work was completed.

These semantic differences stem from the variations in expected usage of each construct.

EXCEPTION HANDLING

PARALLEL ACTIVITY

WF users can place TryCatch activities in their workflows to model the handling of exceptions.

If an exception is thrown but not handled within a workflow, then the host of the workflow decides what action to take. For example, WorkflowApplication can be configured to abort the workflow instance, cancel, or terminate it when an unhandled exception occurs, as shown here:

```
WorkflowApplication instance = new WorkflowApplication(/*faulty workflow*/);
ManualResetEventSlim mre = new ManualResetEventSlim(false);

instance.OnUnhandledException = delegate(WorkflowApplicationUnhandledExceptionEventArgs e)
        {
                Console.WriteLine(e.UnhandledException.Message);
                mre.Set();
                return UnhandledExceptionAction.Terminate;
        };
instance.Run();
//wait for the unhandled event handler to be invoked
mre.Wait();
```

SYSTEM.THREADING.TASKS.PARALLEL.INVOKE

If an unhandled exception is raised by one of the Parallel.Invoke actions then:

1. All the other actions will continue to be executed.
2. An AggregatException containing all the exceptions raised by the faulty actions will be generated.

```
Action[] actions = new Action[]
{
        ()=>{collection.TryAdd("1");},
        ()=>{
                throw new ApplicationException("Test Exception");
```

```
        },
        ()=>{collection.TryAdd("2");}
};
try
{
        Parallel.Invoke(actions );
}
catch (AggregateException  e)
{
        //do something with e
}
```

WORKFLOW PARALLELFOREACH<T> ACTIVITY VS. SYSTEM.THREADING.TASKS.PARALLELFOREACH<T>

DESCRIPTION

PARALLELFOREACH<T> ACTIVITY

As with a foreach loop in C# or a For Each loop in Visual Basic, the WF4 ParallelForEach<T> activity will invoke its child activity for each of the elements in an IEnumerable<T> data source. Like the Parallel activity, the ParallelForEach<T> activity is executed by the WF runtime using a single thread, the execution order of the embedded statements is not defined, and it will not be considered complete until the body is invoked for every element in the data source or the loop is cancelled. If all the child activities are non-blocking, the behavior of ParallelForEach<T> will be the same with the sequential ForEach<T> activity.

A simple sample of ParallelForEach behavior is illustrated below.

```
public static Activity CreateWorkflow()
        {
            var iterationVariable = new DelegateInArgument<int>();

            var workflow = new ParallelForEach<int>
            {
                Values = new InArgument<IEnumerable<int>>((env) => new int[] { 1, 2, 3 }),

                Body = new ActivityAction<int>()
                {
                    Argument = iterationVariable,
                    Handler = new Sequence
                    {
                        Activities =
                          {
                                 new WriteLine { Text = new InArgument<string>(ctx =>
                                 string.Concat ("Text :",
                                 iterationVariable.Get(ctx).ToString(), "
                                 ManagedThreadId:",
                                 Thread.CurrentThread.ManagedThreadId.ToString())) }
                           }
                    }
                }
            };
            return workflow;
        }
WorkflowInvoker.Invoke(CreateWorkflow());
```

SYSTEM.THREADING.TASKS.PARALLEL.FOREACH<T>

The System.Threading.Tasks. Parallel.ForEach<T> has the job of invoking its defined body for each element in the data source. The difference between it and the ParallelForEach<T> activity is that the execution in this case is multi-threaded.

The number of threads used for the execution is controlled by the Parallel.ForEach<T> and ThreadPool's thread injection logic. However, as with Parallel.Invoke and in the same manner, the user can control settings such as the maximum degree of parallelism through a ParallelOptions instance. This includes a maximum degree of parallelism, a target execution scheduler to use

CANCELLATION

PARALLELFOREACH<T> ACTIVITY

As with the cancellation of a Parallel activity, the cancellation of a ParallelForEach activity can be accomplished:

1. At the activity level by using the CompletionCondition property of the ParallelForEach activity.
2. At the global level by canceling the WorkflowApplication host running the activity.

SYSTEM.THREADING.TASKS.PARALLEL.FOREACH<T>

A Parallel.ForEach<T> can be canceled (stopped) in three ways.

1. Using a CancellationToken through ParallelOptions. This is the recommended way to cancel a loop when the desire to stop processing comes from a source external to the loop itself (e.g. a cancel button clicked by a user). This is the same as for Parallel.Invoke; no more details will be added here.
2. Throwing an exception from the body of the loop. This is not recommended as a way to proactively terminate processing of a loop, just as throwing an exception from a sequential foreach or For Each loop is not the recommended way to break out of a loop.
3. Using ParallelLoopState.Break or ParallelLoopState.Stop. This is the recommended way to break out of a loop when the desire to stop processing comes from within the loop body itself.

ParallelLoopState.Break

When the Break method is invoked, the parallel loop will cease execution of the loop at the system's earliest convenience beyond the current iteration. Upon all work quiescing, the ForEach method will return, and the returned ParallelLoopResult structure will contain details about why the loop did not complete all iterations. The semantics of Break are similar to that of the "break" keyword in C#, in that all iterations prior to the one that invoked Break will be completed. As this is a parallel invocation, however, some iterations after the one that invoked Break may have also completed.

```
IEnumerable<int> ds = Enumerable.Range(0, sourceCount);
ParallelLoopResult plr = ParallelLooParallel.ForEach<int>(ds, (step, loopState) =>
{
  if (step == 50)
  {
      loopState.Break();
  }
```

```
  //some work
});
```

ParallelLoopState.Stop

The Stop method results in behavior like that of Break, except that no guarantees are made about what iterations besides the current one have completed.

```
IEnumerable<int> ds = Enumerable.Range(0, sourceCount);
ParallelLoopResult plr = ParallelLooParallel.ForEach<int>(ds, (step, loopState) =>
{
    if (step == 50)
    {
      loopState.Stop();
     }
     //some work
 });
```

EXCEPTION HANDLING

PARALLELFOREACH<T> ACTIVITY

The exception handling behavior for a ParallelForEach<T> activity is identical to that of the Parallel activity.

SYSTEM.THREADING.TASKS.PARALLEL.FOREACH<T>

If an unhandled exception is raised by the body of the loop:

1. The loop will exit as soon as possible. It will not terminate other iterations that are concurrently running, though those iterations have the opportunity to observe that iteration has ended exceptionally and may choose to cooperatively exit early. Once all concurrently running iterations have completed, the loop will exit.
2. An AggregateException containing all the exceptions raised will be thrown out of the ForEach invocation.

USING TPL TASK TO IMPLEMENT WF ASYNCCODEACTIVITY

WF4 provides AsyncCodeActivity and AsyncCodeActivity<TResult> activity base classes in order to support the asynchronous execution of work in a way that does not block the thread being used by the WF runtime to execute the activities in a workflow instance. Use of asynchronous activities in different branches of a Parallel activity allows you to achieve multi-threaded parallelism in a workflow.

When an asynchronous activity executes, the WF runtime automatically considers the workflow instance in a “no persist” zone; in other words, persistence cannot happen while the asynchronous work is pending.

Because the asynchronous activities follow the standard .NET Asynchronous Programming Model (APM), it is natural to integrate with the Task Parallel Library (TPL) and, specifically, with a Task or Task<TResult>.

Below is a simple example of an AsyncCompress activity implemented using TPL Task.

```
public sealed class AsyncCompress : AsyncCodeActivity<Byte[]>
{
    public AsyncCompress()
        : base()
    {
    }
    [RequiredArgument]
    [DefaultValue(new byte[0])]
    public InArgument<byte[]> Data
    {
        get;
        set;
    }

    protected override IAsyncResult BeginExecute(
         AsyncCodeActivityContext context, AsyncCallback callback, object state)
    {
        var data = Data.Get(context);
        if (data == null) throw new ArgumentNullException("data");

        Task<Byte[]> worker = Task.Factory.StartNew((o) =>
            {
                Console.WriteLine("Compressing {0} bytes on thread {1} at {2}",
                        data.Length, Thread.CurrentThread.ManagedThreadId,
                        DateTime.Now.ToString("HH:mm:ss.fff"));

                // Compress the data to an in-memory stream
                var ms = new  MemoryStream(data.Length);
                using (DeflateStream ds = new DeflateStream(ms,
                                                CompressionMode.Compress, true))
                {
                    ds.Write(data, 0, data.Length);
                }
                return ms.ToArray();

            }, state);

        worker.ContinueWith(task => callback(task));
        return worker;
    }

    protected override Byte[] EndExecute(AsyncCodeActivityContext context, IAsyncResult
result)
    {
        var worker = result as Task<Byte[]>;
        if (worker == null) throw new ArgumentException("result");
        return worker.Result;
    }
}
```

Since the Task class implements the IAsyncResult interface (and Task<TResult> derives from Task), there is no additional work needed to create the IAsyncResult. We can simply wrap all the work which needs to be processed asynchronously into a Task and then return that task object from the BeginExecute method. The EndExecute method retrieves the Task object from the supplied IAsyncResult, and returns back that Task's Result.

As is defined by the APM pattern, WF requires that the IAsyncResult object returned from the BeginExecute method must have its AsyncState property set to return the same 'state' object as was supplied to BeginExecute by the WorkFlow runtime in the state parameter. To cope with this, the StartNew (as well as Task's constructor) have overloads to support state objects. Any supplied state will be passed into the Task's delegate and will also be available from the Task's AsyncState property. This is why, in the above example, you see the state object being supplied to the Task.Factory.StartNew method.

Additionally, and again as defined by the APM pattern, WF relies on the AsyncCallback supplied to BeginExecute being invoked once the IAsyncResult instance finishes its execution (e.g. when the Task completes). This can also be easily achieved by adding a continuation to the Task object. This continuation will be scheduled when the Task completes and simply invokes the callback, passing in the Task object as the IAsyncResult.

CANCELLATION

As in other scenarios, an asynchronous activity can be cancelled during the course of workflow execution.

At the cancellation time we also could cancel the worker task, using a CancellationTokenSource passed to the task. In this way, if the activity is cancelled, an attempt to cancel the worker task is made.

For doing this we can make use of the AsyncCodeActivityContext and attach to its UserState the CancellationTokenSource so that we might be able to retrieve later. As a quick demonstration, we modify the BeginExecute() and EndExecute() of the AsyncCompress activity to be

```
protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context,
AsyncCallback callback, object state)
 {
     var data = Data.Get(context);
     if (data == null) throw new ArgumentNullException("data");
     CancellationTokenSource cts = new CancellationTokenSource();
     context.UserState = cts;
     Task<Byte[]> worker = new TaskFactory (cts.Token).StartNew((o) =>
         {
                 Console.WriteLine("Compressing {0} bytes on thread {1} at {2}",
                         data.Length, Thread.CurrentThread.ManagedThreadId,
                         DateTime.Now.ToString("HH:mm:ss.fff"));

                 // Compress the data to an in-memory stream
                 var ms = new  MemoryStream(data.Length);
                 using (DeflateStream ds = new DeflateStream(ms, CompressionMode.Compress,
true))
                 {
                     // before during the real work, make sure it is not cancel-requested
                     if (!cts.Token.IsCancellationRequested)
                                 ds.Write(data, 0, data.Length);
                 }

                 return ms.ToArray();
         }, state);

     worker.ContinueWith(task => callback(task));

    return worker;
 }
```

Then we add support for Cancel() as below

```
 protected override void Cancel(AsyncCodeActivityContext context)
 {
     CancellationTokenSource cts = context.UserState as CancellationTokenSource;
     cts.Cancel();

     base.Cancel(context);
 }
```

This way, when WF cancels this AsyncCompress activity instance for any reason, we can take care of the cancellation of the worker TPL task correspondingly.

EXCEPTION HANDLING

There are two ways that we can handle the exceptions in a scenario like above:

1. Wrapping the Compression algorithm in a try/catch.
2. Benefit from the TPL's built-in ability to catch and prorogate any unhandled exception. Let's say your compression algorithms above encounter an error during execution, the worker TPL task will catch it, save it internally, and throw it back, under a wrapper of an AggregateException, to the WF runtime upon the call of worker.Result inside EndExcute(). The exception can be caught and processed in this moment.

If the last approach is used it is worth mentioning that:

a. If the compression algorithm throws and exception and
b. If the workflow is cancelled before the EndExecute is called

The faulted worker task will not have its exception observed therefore its exception will be re-thrown on the finalizer thread. This situation can be avoided if we added the worker to the user state and during the Cancel call we register a continuation task that will observe its exception, like below:

```
((Task)context.UserState).ContinueWith((task) =>{if(task.IsFaulted) { Exception ex =
task.Exception;}});
```

USING ASYNC ACTIVITIES INSIDE A PARALLEL ACTIVIY

In order to achieve multi-threaded parallelism in WF4, we just need to compose a Parallel / ParallelForEach activity with child activities that are asynchronous activities. Using the same AsyncCompress sample as a showcase,

```
DelegateInArgument<int> iterationVariable = new DelegateInArgument<int> ();
Variable<byte[]> compressed = new Variable<byte[]> ();
var workflow = new ParallelForEach<int>
        {
                Values = new InArgument<IEnumerable<int>>(env => Enumerable.Range(1, 10).Select
                (i => i * 1000)),
               Body = new ActivityAction<int>()
               {
                   Argument = iterationVariable,
                   Handler = new Sequence
                   {
                       Variables = {compressed},
                       Activities =
                       {
                           new AsyncCompress
                           {
                                Data = new InArgument<byte[]>(env =>Enumerable.Range (0,
                                iterationVariable.Get(env)).Select (i => (byte)(i %
                                256)).ToArray()),
                                Result = new OutArgument<byte[]> (compressed)
                           },

                           new WriteLine
                           {
                                Text = new InArgument<string>(env =>string.Format("for input data
                                of size {0}, compressed size is {1} ",
                                iterationVariable.Get(env), compressed.Get(env).Length))
                           }
                       }
```

```
                    }
                }
            };

        WorkflowInvoker.Invoke (workflow);
```

When executing the above workflow, the outputs will be such like

```
Compressing 4000 bytes on thread 7 at 13:59:31.394
Compressing 8000 bytes on thread 5 at 13:59:31.394
Compressing 2000 bytes on thread 5 at 13:59:31.429
Compressing 1000 bytes on thread 7 at 13:59:31.429
Compressing 9000 bytes on thread 4 at 13:59:31.394
Compressing 6000 bytes on thread 8 at 13:59:31.391
Compressing 5000 bytes on thread 9 at 13:59:31.392
Compressing 10000 bytes on thread 3 at 13:59:31.391
Compressing 3000 bytes on thread 10 at 13:59:31.394
Compressing 7000 bytes on thread 6 at 13:59:31.391
for input random data of size 4000, compressed size is 541
for input random data of size 8000, compressed size is 583
for input random data of size 2000, compressed size is 519
for input random data of size 1000, compressed size is 508
for input random data of size 9000, compressed size is 596
for input random data of size 6000, compressed size is 563
for input random data of size 5000, compressed size is 552
for input random data of size 10000, compressed size is 608
for input random data of size 3000, compressed size is 530
for input random data of size 7000, compressed size is 573
```

INTEGRATION OF WF INVOCATION INTO TASK BASED APPLICATION

As easy as it is to apply Parallel Extensions APIs to WF scenarios, it turns out also to be straightforward to integrate WF into a PFX Task based application while using WorkflowInvoker to host workflows.

CONVERTING WORKFLOW INTO TASK

Since WorkflowInvoker provides the standard pair of BeginInvoke() and EndInvoke()methods, we can wrap its asynchronous invocation into a TPL Task utilizing the helper FromAsync API available from TaskFactory.

```
WorkflowInvoker invoker = new WorkflowInvoker(activity);
Task workflowTask = Task.Factory.FromAsync<IDictionary<string, object>>(invoker.BeginInvoke,
invoker.EndInvoke, state /* optional application-specific object */));
```

CANCEL WORKFLOW INSTANCES THROUGH CANCELLATIONTOKEN REGISTRATION

As specified above, Cancellation is a main feature for workflow applications. When dealing with multiple workflow instances invoked in parallel, it might be needed to cancel one (or more instances at once). The CancellationToken.Register offers great support for this type of scenarios as shown below.

```
var workflow = new Parallel
        {
```

```
            Branches =
            {
                new Sequence(){
                    Activities=
                    {
                        new Delay(){ Duration= TimeSpan.FromSeconds(5)},
                        new WriteLine(){ Text="Second:5 seconds delay." }
                    }
                },
                new WriteLine(){ Text="First }
            }
        };

    CancellationTokenSource cts = new CancellationTokenSource();
    Task workflowTask1 = Task.Factory.StartNew( ()=>
        {
            ManualResetEventSlim wkfEnd = new ManualResetEventSlim(false);
            Console.WriteLine("Workflow1 execution will start);
            WorkflowApplication wkfApp = new WorkflowApplication(workflow);
            cts.Token.Register( ()=>{wkfApp.Cancel();});
            wkfApp.Completed = (workflowCompletedEvArgs) => { wkfEnd.Set(); };
            wkfApp.Run();

            wkfEnd.Wait();
        }, cts.Token);

    Task workflowTask2 = Task.Factory.StartNew(() =>
    {
        ManualResetEventSlim wkfEnd = new ManualResetEventSlim(false);
        Console.WriteLine("Workflow2 execution will start."
        WorkflowApplication wkfApp = new WorkflowApplication(workflow);
        cts.Token.Register(() => { wkfApp.Cancel(); });
        wkfApp.Completed = (workflowCompletedEvArgs) => { wkfEnd.Set(); };
        wkfApp.Run();

        wkfEnd.Wait();
    }, cts.Token);

    //some more work
    Thread.Sleep(2000);
    //cancel the workflows' execution
    Console.WriteLine("The workflows will be cancelled.");
    cts.Cancel();

    //wait for the workflow tasks to finish
    Task.WaitAll(workflowTask1, workflowTask2);
```

INVOKE ACTIVITY UNDER SYSTEM.THREADING.TASKS.PARALLEL AND ANY TASK APIS

Since a workflow activity is independently executed on a single thread, we can safely invoke a collection of such activities under the control of System.Threading.Task.Parallel.* APIs.

```
System.Threading.Tasks.Parallel.ForEach(IEnumerable<Activity>, (activity) =>
  {
       Console.WriteLine("Executing workflow {0} on thread {1} at {2}",
        activity.DisplayName, Thread.CurrentThread.ManagedThreadId,
        DateTime.Now.ToString("HH:mm:ss.fff"));

       WorkflowInvoker.Invoke(activity);
  });
```

Workflow invocation could also be integrated into any task APIs, as shown below, which will trigger the workflow execution only when a previous workflow completes.

```
System.Threading.Tasks trigger = Task.Factory.StartNew(() => {
WorkflowInvoker.Invoke(previous_activity); });
Trigger.ContinueWith(t => { WorkflowInvoker.Invoke(next_activity); });
```

SAMPLES OF CUSTOM ACTIVITIES

TASKPARALLELINVOKE ACTIVITY

If a number Actions need to be executed in parallel inside a workflow instance, the bellow TaskParallelInvoke activity shows how this scenario can be achieved using System.Threading.Tasks.Task and System.Threading.Tasks .Parallel.Invoke.
If integrated in a workflow like below:

```
<Parallel>
        <A />
        <TaskParallelInvoke>
                LongRunningAction1
                LongRunningAction2
        </ TaskParallelInvoke >
        <B />
        <C />
        <D />
</ Parallel >
```

the TaskParallelInvoke activity:

1. Does not block the single workflow thread from executing the B,C,D activities
2. Can be cancelled independently without canceling the whole workflow, through the CancellationTokenSource.

The WF runtime guarantees that during the execution of an AsyncCodeActivity, persistence of the workflow instance is not allowed.

```
/// <summary>
/// ParallelInvocation of the value actions
/// </summary>
public sealed class TaskParallelInvoke : AsyncCodeActivity
{
    public TaskParallelInvoke()
        : base()
    {
    }

    [RequiredArgument()]
    [DefaultValue(null)]
    public InArgument<CancellationToken> CancellationToken
    {
        get;
        set;
    }
```

```
[RequiredArgument]
[DefaultValue(null)]
public InArgument<Action[]> Values
{
    get;
    set;
}

[RequiredArgument]
[DefaultValue(-1)]
public InArgument<int> MaxDegreeOfParallelism
{
    get;
    set;
}

protected override void CacheMetadata(CodeActivityMetadata metadata)
{
    Collection<RuntimeArgument> args = new Collection<RuntimeArgument>();

    RuntimeArgument valuesArgument = new RuntimeArgument("Values", typeof(Action[]),
    ArgumentDirection.In, true);
    metadata.Bind(this.Values, valuesArgument);
    args.Add(valuesArgument);

    RuntimeArgument tokenArgument = new RuntimeArgument("CancellationToken",
    typeof(CancellationToken), ArgumentDirection.In, true);
    metadata.Bind(this.CancellationToken, tokenArgument);
    args.Add(tokenArgument);

    RuntimeArgument maxDegreeOfParallelismArgument = new RuntimeArgument("MaxDegreeOfParallelism",
    typeof(int), ArgumentDirection.In, true);
    metadata.Bind(this.MaxDegreeOfParallelism, maxDegreeOfParallelismArgument);
    args.Add(maxDegreeOfParallelismArgument);

    metadata.SetArgumentsCollection(args);
}

//cancel the application token source
protected override void Cancel(AsyncCodeActivityContext context)
{
    CancellationTokenSource applicationTokenSource = (CancellationTokenSource)context.UserState;
    applicationTokenSource.Cancel();
    base.Cancel(context);
}

protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context, AsyncCallback
callback, object state)
{
    Tasks.ParallelOptions options = new Tasks.ParallelOptions();

    //the activity has its own token that the user can cancel from outside
    //however if the activity needs to be cancelled by a workflow application - we will link the
    token with a global one
    //when any of them will be cancelled the parallel invocation will be cancelled.
    CancellationTokenSource applicationTokenSource = new CancellationTokenSource();
    CancellationToken applicationToken = applicationTokenSource.Token;
    context.UserState = applicationTokenSource;

    if (this.CancellationToken != null)
    {
        //link the two tokens
        CancellationTokenSource cts = CancellationTokenSource.CreateLinkedTokenSource(
        applicationToken,
        this.CancellationToken.Get(context));
        options.CancellationToken = cts.Token;
    }
```

```
        options.MaxDegreeOfParallelism = this.MaxDegreeOfParallelism.Get(context);

        //run ParallelInvoke async
        Action[] values = this.Values.Get(context);
        System.Threading.Tasks.Task parallel = Tasks.Task.Factory.StartNew((taskState) =>
            {
                try
                {
                    Tasks.Parallel.Invoke(options, values);
                }
                catch(OperationCanceledException )
                {
                    //expected if the token was cancelled
                }
            }, state);

        //force the EndExecute
        parallel.ContinueWith((task) => { callback.Invoke(task); });

        return parallel;
    }

    protected override void EndExecute(AsyncCodeActivityContext context, IAsyncResult result)
    {

    }
}
```

TASK ACTIVITY AND TASKWAIT ACTIVITY

In certain scenarios, it might be beneficial to launch an arbitrary child activity (most likely a complex but independent one) in an asynchronous fashion, so that the current thread serving the overall workflow instance will not be blocked until the result from the sub activity fired earlier is absolutely required. The following sample that should be used in a no-persit zone demonstrates how this can be easily achieved via integrating with TPL Task.

```
/// <summary>
/// An activity which launches any arbitrary activity delegate in a TPL Task.
/// It returns a TPL Task object which could be given to the WaitTaskActivity at later time
/// </summary>
/// <typeparam name="T"></typeparam>
class TaskActivity<T> : CodeActivity<Task>
{
    /// <summary>
    /// Any argument to be given to the Handler
    /// </summary>
    public InArgument<T> Argument
    {
        get;
        set;
    }

    /// <summary>
    /// The arbitrary activity delegate to be executed
    /// </summary>
    public ActivityAction<T> Handler
    {
        get;
        set;
    }

    /// <summary>
    /// Run the arbitrary activity delegate in a TPL Task on background thread
    /// </summary>
    /// <param name="context"></param>
```

```
        /// <returns></returns>
        protected override Task Execute(CodeActivityContext context)
        {
            T arg = context.GetValue(Argument);
            return Task.Factory.StartNew(() =>
                {
                    var workflow = new Sequence
                           {
                               Activities =
                                 {
                                    new InvokeDelegate
                                    {
                                        Delegate = Handler,
                                        DelegateArguments =
                                        {
                                            { "Argument", InArgument<T>.FromValue (arg) }
                                        }
                                    }
                                 }
                           };

                    WorkflowInvoker.Invoke(workflow);
                });
        }
    }

    /// <summary>
    /// An activity to ensure an input TPL Task reach its Completed state
    /// </summary>
    class WaitTaskActivity : CodeActivity
    {
        /// <summary>
        /// The input TPL Task
        /// </summary>
        [RequiredArgument]
        public InArgument<Task> Task
        {
            get;
            set;
        }

        /// <summary>
        /// Block the current thread until the input TPL Task is Completed
        /// </summary>
        /// <param name="context"></param>
        protected override void Execute(CodeActivityContext context)
        {
            Task task = context.GetValue(Task);
            task.Wait();
        }
    }
```

With the TaskActivity and WaitTaskActivity above, it is fairly easy to build up composite workflow which has the ability to schedule any activity delegate asynchronously, as shown below

```
            Variable<Task> task = new Variable<Task>();
            var delegateVariable= new DelegateInArgument<int>();

            var workflow = new Sequence
            {
                Variables = { task },
                Activities =
                {
                    new TaskActivity<int>
                    {
                         Argument = InArgument<int>.FromValue (10),
                         Handler = new ActivityAction<int>()
                           {
```

```
                    Argument = delegateVariable,
                    // all Activities under this Handler are executed on a separate thread
                    Handler  = new Sequence
                    {
                      Activities =
                      {
                         new WriteLine
                         {
                             Text = ExpressionServices.Convert<string>(ctx =>
                                    string.Format ("Task executing in Thread {0}, Input {1} at
                                    {2}",
                                    Thread.CurrentThread.ManagedThreadId,
                                    delegateVariable.Get (ctx),
                                    DateTime.Now.ToString("HH:mm:ss.fff")))
                         }

                      }
                    }
                } ,

            Result = OutArgument<Task>.FromVariable (task),
        },

        // perform any other activities on the current thread
        // while the background thread runs the above TaskActivity
        new WriteLine
        {
            Text = ExpressionServices.Convert<string>(ctx => string.Format ("Checking at
            {0}", DateTime.Now.ToString("HH:mm:ss.fff")))
        },

        // now it's time to ensure the async TaskActivity is completed
        new WaitTaskActivity
        {
            Task =  InArgument<Task>.FromVariable(task)
        },

        new WriteLine
        {
            Text = ExpressionServices.Convert<string>(ctx => string.Format ("Task Done at
            0}",DateTime.Now.ToString("HH:mm:ss.fff")))
        }
    }
};

WorkflowInvoker.Invoke(workflow);
```

CONCLUSIONS

To summarize, the two similarly named features delivered in .NET 4, Parallel/ParallelForEach Activity in Windows Workflow Foundation and Parallel/ParallelForEach APIs as part of Parallel Extensions Framework are targeted at different customers with their own distinctive scenarios. The Parallel WF activity is designed for easy composition with other workflow activities and it achieves interleaved parallel execution of child activities; the Parallel APIs is more suitable for any general audience who wants to schedule some asynchronous work on certain threadpool background thread. However, despite their distinctiveness, those two technologies can be integrated together to create more powerful workflow applications. As have been illustrated above, customized activities can be developed by simply plugging in various library APIs from Parallel Extensions Framework into Workflow infrastructure to accomplish multi-threaded parallelism while still keep the easy composability.

KEY TAKEAWAYS

1. Consider Parallel Extensions to the .Net 4, APIs for enabling multi-threaded workflow scenarios.
   System.Threading.Tasks.Parallel. and System.Threading.Tasks APIs can easily be integrated into the workflow scenarios. Highly scalable across the machine CPUs, the Parallel.* and Tasks APIs have built in support for waiting, continuation, cancellation, exception handling, etc.*

2. Consider adopting the CancellationTokenSource /Cancellation Token for fine granular cancellation of workflow activities.
   Light weight types CancellationTokenSource and CancellationToken represent a nice fit for supporting the cancellation of long running Workflow Activities.

3. Consider using the new thread safe collections and synchronizations primitives if needed, when using workflow multi-thread scenarios.

ACKNOWLEDGEMENTS

The authors would like to thank Bob Schmidt and, Stephen Toub for their great feedback on drafts of this paper.

REFERENCES

1. http://msdn.microsoft.com/en-us/library/ee342461.aspx
2. http://msdnstage.redmond.corp.microsoft.com/en-us/library/ee839372(VS.100).aspx
3. http://msdnstage.redmond.corp.microsoft.com/en-us/library/system.activities.statements(VS.100).aspx
4. http://msdn.microsoft.com/en-us/library/dd647810(VS.100).aspx
5. http://msdn.microsoft.com/en-us/library/system.activities.statements.parallel(VS.100).aspx
6. http://www.danielmoth.com/Blog/2009/01/parallelising-loops-in-net-4.html
7. http://blogs.msdn.com/pfxteam/archive/2009/05/22/9635790.aspx